From DSL Specification to Interactive Computer Programming Environment

Pierre Jeanjean, Benoit Combemale, Olivier Barais

From DSL Specification to Interactive Computer Programming Environment

Pierre Jeanjean
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
pierre.jeanjean@inria.fr

Benoit Combemale
University of Toulouse
Toulouse, France
benoit.combemale@irit.fr

Olivier Barais
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
olivier.barais@irisa.fr

Abstract

The adoption of Domain-Specific Languages (DSLs) relies on the capacity of language workbenches to automate the development of advanced and customized environments. While DSLs are usually well tailored for the main scenarios, the cost of developing mature tools prevents the ability to develop additional capabilities for alternative scenarios targeting specific tasks (e.g., API testing) or stakeholders (e.g., education). In this paper, we propose an approach to automatically generate interactive computer programming environments from existing specifications of textual interpreted DSLs. The approach provides abstractions to complement the DSL specification, and combines static analysis and language transformations to automate the transformation of the language syntax, the execution state and the execution semantics. We evaluate the approach over a representative set of DSLs, and demonstrate the ability to automatically transform a textual syntax to load partial programs limited to a single statement, and to derive a Read-Eval-Print-Loop (REPL) from the specification of a language interpreter.

CCS Concepts • **Software and its engineering → Domain specific languages**.

Keywords repl, language engineering, domain specific languages

1 Introduction

Domain-Specific Languages (DSLs) are software languages specifically tailored (e.g., syntax, semantics and tools) for a given problem space. While attractive in many application domains, the success of DSLs relies on the ability of language workbenches to support their specifications and automate the development of advanced and customized tools. In the last decade, various language workbenches have been proposed, with different facilities, formalisms and underlying implementation frameworks [5].

While covering together a broad range of programming and modeling facilities, language workbenches face interoperability issues [4]. In particular language workbenches are usually well equipped in terms of formalisms, implementation frameworks and generative approaches to eventually obtain a fine-tuned specific set of tools, initially chosen for targeting the main scenarios. But then, it often fails to encompass other forms of implementation that would fit alternative scenarios. DSL specifications are hardly reusable for driving the development of various forms of DSL environments [1].

For example, while nowadays comprehensive BNF-based interpreted language specifications would drive the development of comprehensive DSL environments, it is hard to drive the development of alternate environments such as interactive computer programming environments. They are interactive environments that help to document (*e.g.,* Jupyter[1] [14]), to evaluate APIs ([13, 17]), to teach [2, 6, 7], to share knowledge (*e.g.,* in science [9, 11]), or for arts [15]. In practice, most of today's DSL specifications have a single execution entry-point responsible for executing the whole program. Interactive environments support multiple entry points, with different strategies for managing the execution context and flow. Beyond the parser itself, the execution engine must turn into a *Read-Eval-Print Loop* (REPL), that repeatedly reads an input from the user, evaluates it and prints a result. One of the main advantages of using a development environment that includes a REPL is to be able to write and execute a program piece by piece.

In this paper, we propose an approach to automatically generate interactive computer programming environments from existing specifications of textual interpreted DSLs. More specifically, the approach reifies the relevant abstractions to use when specifying the DSL, and use a combination of static analysis and language transformations to automate the translation of both the language syntax and semantics. The syntax is converted such as it is possible to load partial program limited to single statements, and the semantics in the form of a language interpreter is turned into a REPL. Hence, we provide the required generative approach to automate the integration into a ready-to-use interactive computer programming environments in the form of both a language shell into Eclipse, and a specific Notebook (*a la* Jupyter).

We provide an implementation of the approach as a prototype on top of the Eclipse Modeling Framework [16], which includes Ecore[2] for the syntax definition, Xtext[3] for the textual syntax definition, and ALE[4] for the operational semantics definition. The approach would be similarly applied to other alternatives within EMF (*e.g.,* Java or ATL interpreters,

[1] cf. https://jupyter.org/
[2] cf. https://www.eclipse.org/modeling/emf/
[3] cf. https://www.eclipse.org/Xtext
[4] cf. http://gemoc.org/ale-lang

or using the native EMF Switch), or other implementation frameworks (*e.g.*, Truffle-based interpreter). The approach applies to any DSL specification with a BNF definition of the syntax which includes an explicit definition of the execution flow, and an operational semantics specification in the form of a well-defined (top-down) interpreter or visitor.

We apply our approach on several DSLs, namely *Logo*, *MiniJava* and *ThingML* [8]. According to the different paradigms (*e.g.*, control-flow vs. data-flow), we discuss the different ways to operate our approach and provide relevant interactive computer programming environments.

Hence, we provide a well-defined and automatic approach for complementing a modeling environment with an interactive computer programming environment. The resulting environment also provides several extension points to possibly customize the execution flow. Finally, we discuss several perspectives, including the deployment on web technologies.

The remainder of this paper is structured as follows. We motivate our contribution in Section 2, and introduce an illustrative example used throughout the paper. Section 3 presents an overview of our approach, while Section 4 gives the details of the transformation of the syntax and the semantics. We present an evaluation of our approach in Section 5, and discuss related works in Section 6. Finally, a conclusion and future research directions are presented in Section 7.

2 Background and Motivation

The main objective of our approach is to automatically generate an interactive computer programming environment from a DSL specification initially designed to drive the development of a comprehensive DSL environment with an advanced editor, a regular interpreter, etc. Such a specification usually involves a syntax definition, e.g., in the form of a BNF-like description, and a definition of the semantics, e.g., an operational semantics in the form of an interpreter (or any variant such as a visitor pattern). A language definition also includes static semantics, which define all the context conditions that ensure statically correct conforming programs. In such a definition, the language usually encompasses a single execution entry point, a finely tuned execution context and an interpreter which defines a particular traversal of a given syntax tree to manipulate and update the execution context over the execution.

Such a language definition is now well supported by advanced language workbenches that help language engineers to develop language tools such as structured editors, debuggers and simulators. For instance, tools like Xtext[5] support the generation of an advanced editor with a parser, syntactic validation, and all the features of modern code editors (*e.g.,* syntax coloring, auto completion, etc.). The GEMOC Studio[6] helps to complement the language tooling with advanced execution engines and debugging facilities. Other language workbenches (*e.g.,* MPS, Rascal, Spoofax) offer similar facilities [5]. Most of these workbenches are now mature enough for being included in industrial settings, and allow language engineers to automate the development of the tools for the main scenarios of the expected language users.

Let us consider for example the language *Logo*[7]. *Logo* is an educational language whose main focus is the animation of turtle graphics. As such, most of the statements are accompanied by a feedback which is an action from the turtle, and this is part of what makes it interesting for teaching purposes. From a comprehensive definition of the language, it is possible to automatically generate a dedicated and structured editor that supports the definition of complete logo programs, including a functional architecture and an explicit execution flow across the architecture.

However, it has been recently recognized that the different tasks performed with a given language, possibly by different stakeholders, would require specific language support (*e.g.*, dedicated environments with the right facilities) [1]. For instance, while it can be convenient to have a comprehensive editor for editing complex logo programs, one would like access to other kinds of environment such as interactive environments that allow to immediately get the result of the program at the time it is being edited. Such an environment would be very useful for education purposes, such as learning about the language or evaluating existing libraries. Generally speaking, it becomes way easier to introduce a programming language to beginners if they don't have to conform to the structure of an entire program when they want to write their very first instructions [6]. *Java* is a perfect example of this [2, 7], as this is a language that requires to follow a rather complex process to simply print 'Hello World!' in a terminal emulator: the programmer needs to define a public class, declare a method both public and static and give it a specific name, while also specifying that it will require a certain type of arguments, before finally writing the very first statement of a program. Of course, the syntax makes perfectly sense, and understanding it will be important in order to learn how to use the language, but it can be detrimental to introduce it during the first glimpse at Java programming. An interactive computer programming environment can be beneficial to simply learn how specific language statements work in practice. In the case of *Logo*, interactive computer programming environments would provide immediate feedback from any statement of the language, and help learning complex concepts. Note that interactive computer programming environments can be also beneficial for experienced language users, in order to test the behavior of code snippets at any time. It makes possible to run processes in an arbitrary order while having access to all the intermediary results, which can be very useful when experiencing a new

[5] cf. https://www.eclipse.org/Xtext/
[6] cf. http://gemoc.org/studio.html
[7] cf. https://en.wikipedia.org/wiki/Logo_(program

API. Beyond learning the different facilities provided by the API, it may also help to learn specific protocols in the use of the API.

Interactive computer programming environment can take various forms, including a basic language shell where single statements can be executed sequentially, based on a global context, and possibly with the history of the intermediate states of the built program. Alternatively, a notebook interface provides a virtual notebook environment used for literate programming. It support the definition of a sequence of pieces of code, with intermediate results, and possibly word processing to document the program.

In all cases, an interactive computer programming environment requires a language interpreter in the form of a read–eval–print loop (REPL), which is able to execute a program piece by piece. This requires to support different execution entry-points, and a specific management of the execution context and flow. The context is usually global (though some scoping rules can still apply), and the execution flow is sequential, starting from the starting point defined by the programmer. Other strategies may exist, in particular regarding the execution flow where a specific order may be imposed to keep reproducible executions.

REPL execution engines offer facilities such as: history of inputs and outputs, input editing and context specific completion over symbols, path names, class names and other objects, as well as help and documentation for commands.

Nowadays, most general purpose languages take these considerations into account, and ship their own REPL implementation: *Python* and *C#* run in interactive mode by default, *Node.js* can execute the "repl" module out of the box, *PHP* can be run as an interactive shell with the switch "-a", *Swift* offers a REPL within Xcode, *Ruby* is shipped with the executable "IRb", and even *Java* includes its own REPL "JShell" since the version 9 of JDK.

While interactive computer programming environments have been specifically developed for general purpose languages, there is currently no approach that helps to turn an existing DSL specification in a way that a complementary interactive environment can be automatically generated. Instead, a new specification must be established for the specific purpose of driving the development of an interactive environment. Hence, the research question (RQ) we address in this paper is the following:

> RQ: Is it possible to automate the transformation of an existing textual and interpreted DSL specification in such a way that an interactive computer programming environment can be automatically derived?

We consider a DSL specification that includes:

- a syntax described as a grammar defined using a rule language based on "Extended Backus-Naur Form Expressions",
- static semantics built from a set of first order logic rules, and
- operational semantics based on a *pure* interpreter pattern. *Pure* means that each *interpret* method of the interpreter accesses only the node object attributes and the context object attributes.

Our approach provides the required abstractions for complementing an existing DSL specification with the minimal information needed for generating a REPL (*i.e.*, the expected execution entry points and their associated documentation and output messages), and automates the transformation of the DSL specification so that an interactive computer programming environment can be derived. In particular, we automate the transformation of the syntax to parse separate pieces of code, and the semantics to support the execution of single statements according to a specific execution context and flow. We also propose a unified REPL interface in order to derive different kinds of environment, e.g., a language shell and a notebook interface.

3 Approach Overview

The main objective of our approach is to automatically transform an existing DSL specification (cf. upper left part in Fig. 1) initially used to drive the development of a comprehensive integrated development environment (cf. lower left part in Fig. 1), into a new one (cf. upper right part in Fig. 1) that can be used to automate the development of interactive computer programming environments (cf. lower right part in Fig. 1).

Since we are aiming for a systematic transformation process, we put some restrictions on the supported DSLs: they need an extended BNF grammar and operational semantics that follow a pure interpreter pattern. We believe that these characteristics are ones of the most common, and thus that these are acceptable limitations.

To reach this objective and address the RQ specified in the previous section, we identified four challenges:

- C1 Identification of the different execution entry points that are meaningful for the corresponding REPL, and the expected outputs and help messages given to the user.
- C2 Transformation of the syntax so that we can parse and load partial programs corresponding to the identified execution entry points.
- C3 Definition of a sound yet flexible execution context and flow management,
- C4 Transformation of the semantics so that we can execute pieces of code corresponding to the identified execution entry points.

Entry-points Identification (C1) We define as language entry-points the constructs that a language user can use and that can be executed outside of any other context. With most traditional DSLs, the execution can only handle a complete

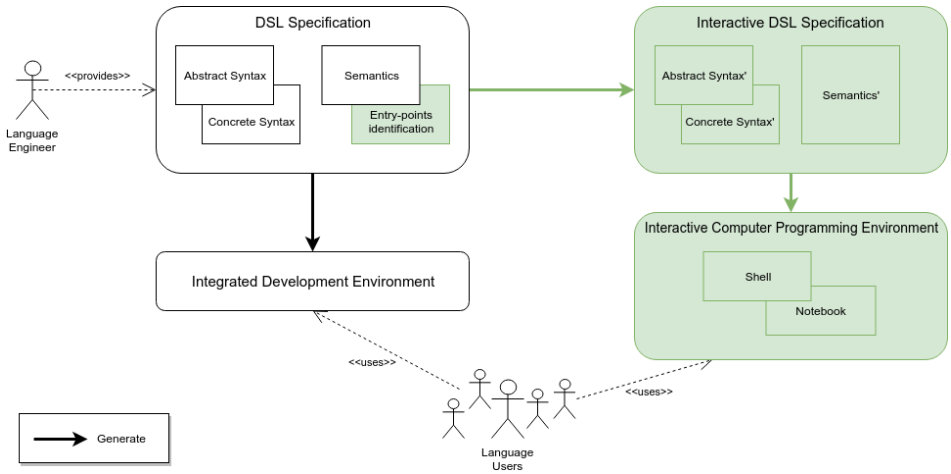

Figure 1. Programming Environment Generation from DSL Specifications

program and builds a context for it, that will later be used by all the statements and expressions. The only entry-point is as such the *complete program*. Here, we want to provide several entry-points with a granularity lower than a regular program.

However, the granularity of the new execution entry-points cannot be inferred automatically. The choice is up to the language engineer. In practice, they correspond to any expression to be considered as an executable statement within the interactive environment. It is therefore necessary to provide within our approach means of specifying these new entry points, and the underlying framework for loading and saving single statements.

To report the execution entry-points, relevant abstractions must be provided to the language engineer for enhancing the initial DSL specification. Abstractions must support the identification of the relevant statements to be executed independently. Moreover, to give intermediate feedback to the language user, the new entry-points need to be also supplied with additional information about the expected outputs (*e.g.*, the user would expect to get an evaluation result when he inputs a *Logo* expression), and possibly an help message.

In addition to the identification of the new execution entry-points in the DSL specification, a corresponding framework must be provided to save and load partial programs corresponding to the possible entry-points. For such a purpose, we transform the syntax specification in order to make partial programs valid for the parser and the corresponding syntax tree. We refer to these partial programs as *instructions*.

Transformation of the Syntax (C2) On the basis of the identified entry-points, the existing syntax specification must be transformed to enable all of these entry-points as valid instructions. A new root rule within the grammar specification and a new root node for the AST named *Interpreter* are integrated. The latter contains all the newly defined valid entry points, and possibly the definitions of additional behaviors to instrument the execution.

Execution Context and Flow Management (C3) We do not handle complete programs anymore but independent instructions. In order to keep a consistent execution through the different iterations of the REPL, a global execution context and its flow along the independently executed instructions must be managed. This is the role of the proposed *Interpreter*, that will instantiate a context then simply pass it to instructions before executing them. Execution results must also be stored in a specific variable, and an execution trace manager must be provided to offer a complete history.

We propose a generic interface to interact with (sequences of) instructions, used by generic interactive computer programming environments such as a language shell and a notebook interface.

Transformation of the Semantics (C4) The last step of our approach consists in transforming the DSL semantics, so that the instructions can be executed independently, over a global context, and according to the proposed interface. In order to automate this transformation, we make several assumptions about the form of the DSL specification. In our current approach, we consider operational semantics defined according to the interpreter design pattern, *i.e.* an operation associated with each node of the AST, and the same context object associated with this operation containing all the dynamic information related to the language semantics. We also assume that the context passed to each nodes can be instantiated and initialized from the *Interpreter* node. If the context cannot be properly initialized on its own, we still give the ability to a language engineer to include custom rules in the semantics, but we do not try to infer them during the REPL language generation. Finally, each operation associated to a node of the syntax tree declared as an entry point must not make assumptions about the execution context

other than that related to the initialization, nor about the structure of the parent nodes. We defined this property as a *pure* interpreter pattern.

4 Technical Details and Implementation

This section describes the technical details of the different steps of our approach, and proposes a particular implementation[8].

The proposed implementation comes in the form of a prototype based on the GEMOC Studio [3]. The GEMOC Studio is an Eclipse package on top of the Eclipse Modeling Framework [16], which has been experienced in various industrial projects. Among others, it offers a language workbench that supports the modular specification of DSLs, using Ecore for the abstract syntax, Xtext for the textual concrete syntax, OCL or Xtend for the static semantics and ALE for operational semantics. Other alternatives are also proposed but not illustrated in the scope of this paper.

We illustrate both the approach and the implementation using the simple but real-world *Logo* language introduced in Section 2.

4.1 DSL Specification Enhancement

As presented in Section 3, we first provide to the language engineer the relevant abstractions for specifying the multiple execution entry-points:

- Identification of the valid instructions to be executed independently,
- Definition of the expected outputs as intermediate results, and
- Definition of the help messages for the language user.

In practice, these information could be provided either in the syntax or in the semantics. However, we had to consider that the visitor can be augmented by additional helpers for a given Ecore object, and there is no way of deciding on which to use. Besides, the output needs to refer to dynamic information which is mainly available within the semantics.

In order to identify the required entry-points, the language designer could methodically:

1. take a look at each rule of the grammar and choose the ones to provide in the REPL
2. decide on the expected outputs for each of the chosen rules
3. factor them in to abstract parent rules if the outputs are the same

To let the language engineer define the required information, we introduce a new metamodel shown in Fig.2. It can be seen as a dedicated meta-language to modularly complement the initial DSL specification with information related to the REPL. The core element of this metamodel is the *Instruction* meta-class. It defines three main information required to generate the REPL:

1. the new entry point in referencing a specific AST and the ALE method that defined the operational semantics for this node.
2. the help message to display if a user wishes to request help on this specific entry point.
3. the elements of the semantics to be used as textual outputs of the interpreter. These elements can be either attributes related to the execution (*i.e.,* attributes of the operational semantics), calls to existing methods of these semantics, or calls to ALE methods defined by the language engineer (*e.g., evalResult* and a set of *evalParams* that could target an *ALE Expression*).

The second main meta-class is *Interpreter*. It allows language engineers to specialize, among other things, the initialization of the execution context of the interpreter.

We provide two concrete syntax to populate the model conform to the *ReplDefinition* metamodel: a dedicated DSL, and additional annotations to ALE. This model represents the required information to drive the transformation of the DSL specification.

Using a New DSL The first concrete syntax is a new DSL built as an extension of ALE. Fig. 3 shows the definition of the REPL for the *Logo* language.

The first part defines the entry points, associated outputs and help messages. It defines two new entry points: Statement::execute and Expression::compute. It also defines their associated outputs to display: *logo_repl.turtle.toString()* and *output.toString()* (*output* refers to the actual value possibly returned by the entry-point). Finally we could define the help associated with these entry points (it has been done only partially in the Logo example).

The second part specifies the *Interpreter*, the specialization of its context of execution and its initialization. *Interpreter* will serve as the starting point of the execution of the REPL and will manage the future instructions. It will contain the same kind of runtime data as the entry-point of the base DSL, which will define the global context of the REPL. In the case of *Logo*, this means the turtle graphics, and a symbol table used to define procedures (using a symbol table for this is simply a design choice of the language engineer). In order to initialize this global context, the initialization method of the interpreter will also be the same as the base DSL.

Using Annotations The same kind of information can be defined directly within the existing ALE operational semantics using a set of annotations. We provided the language engineer with the following annotation:

```
@repl__outputtarget__outputcall__...
```

The output specification here is optional, and represents either an attribute read or a method call on a semantic object

[8] See our prototype at https://anonymous.4open.science/r/84105ce9-f47c-4dd2-936e-9eb2dd345ad0/

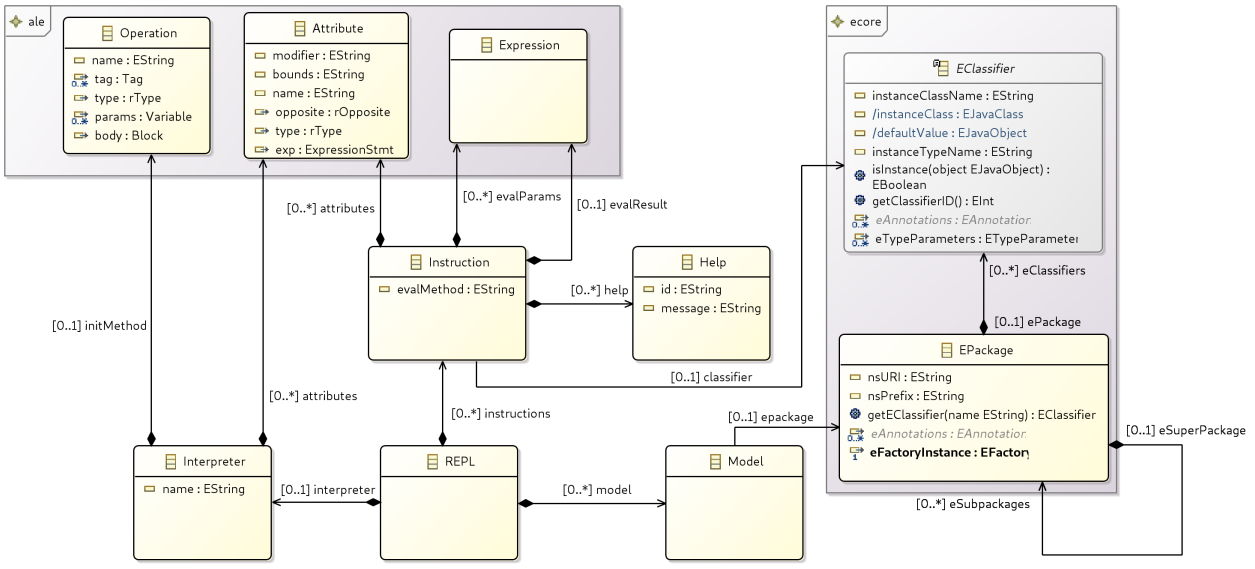

Figure 2. ReplDefinition Metamodel

```
extend http://www.gemoc.org/logo as logo

instruction logo.Statement:
    help right "Turn turtle of 'p' degrees to the right"
    help forward "Move turtle of 'p' units forward"
    execute(logo_interpreter.turtle, logo_interpreter.st)
        => logo_interpreter.turtle.toString();
instruction logo.Expression:
    compute(logo_interpreter.st)
        => output.toString();

interpreter logo_interpreter {
    attribute Turtle turtle;
    attribute SymbolTable st;
    initmethod def void init() {
        self.turtle := Turtle.create();
        self.turtle.xpos := 0.0;
        /* ... */
        self.st := SymbolTable.create();
        self.st.init();
    }
}
```

Figure 3. Example of ReplDefinition Model for Logo

```
open class Statement {
    /**
     * right: Turn turtle of 'p' degrees to the right
     * forward: Move turtle of 'p' units forward
     */

    @repl__turtle__toString
    @step
    def void execute(Turtle turtle, SymbolTable st) {
        'ERROR: Unknown statement'.log();
    }
}

open class Expression {
    @repl__output__toString
    @step
    def Value compute(SymbolTable st) {
        'ERROR: Unknown expression'.log();
        result := null;
    }
}
```

Figure 4. ReplDefinition Annotations Used on Logo

from the global context, or on the result of the operation being annotated. Note that the syntax is based on underscores because of limitations of the ALE language, which only supports identifiers as annotations. Using two underscores as the separator allows for compatibility with semanics using either camel case or snake case for identifiers.

The language engineer can also set the help message to display by using a *javadoc* like comment:

```
/**
 * keyword: Help message
 */
```

In the base semantics for *Logo*, the only additions besides the optional help messages were the two following annotations (cf. Github repository):

- `@repl_turtle_toString` for the `execute` operation of *Statement*
- `@repl_output_toString` for the `compute` operation of *Expression*

Fig 4 shows a code excerpt from the operational semantics of the Logo language extended with the proposed annotations to define the information required to complement the DSL specification with an interactive programming environment. The set of annotations is however less expressive than the DSL. Indeed a language engineer might want to use a

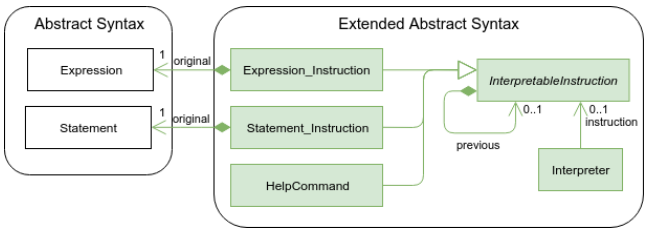

Figure 5. Abstract Syntax Extension for Logo

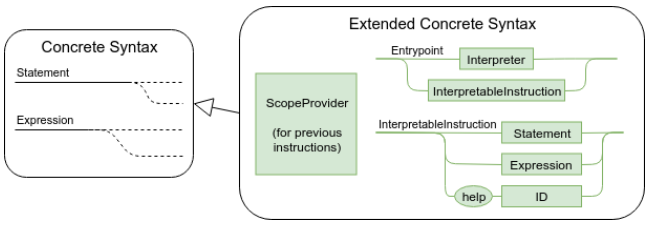

Figure 6. Concrete Syntax Extension for Logo

```
// Import existing Logo definition.

EntryPoint returns ecore::EObject:
  InterpretableInstruction | Interpreter;

InterpretableInstruction:
  {Statement_Instruction} original=Statement
    | {Expression_Instruction} original=Expression
    | {HelpCommand} 'help' command=ID;

Interpreter:
  {Interpreter}
```

Figure 7. Generated Extended Grammar Definition for Logo

more complex expression as the output of an instruction or specialize the execution context for the REPL, which could not be done through annotations. One could still choose to modify the behavior of the base semantics by adding a new ALE operation that could then contain any kind of ALE expression, and call it in the annotation. This new operation could not, however, have access to the global context of the interpreter, nor to the intermediary results.

Based on this information, the DSL specification can be transformed so that a REPL can be derived and used by interactive computer programming environments. It is defined in three steps i) Abstract Syntax Tree transformation, ii) Concrete Syntax transformation, iii) Operational semantics transformation. The next subsections detail these three transformations applied on the original DSL specification. Finally, we present the generic REPL interface provided and the clients defined as interactive computer programming environments: a language shell and a notebook interface.

4.2 Abstract Syntax Transformation

During the DSL specification transformation process, we first complement the abstract syntax with additional concepts.

We first define *Interpreter* as the REPL entry-point. It owns a reference to the abstract class *InterpretableInstruction* which will be set during the execution to always target the current instruction.

InterpretableInstruction also has a containment to itself, which creates a linked list of the previously executed instructions, hence keeping the whole execution history in a single resource. For each instruction *I* defined in the *ReplDefinition* model, the new abstract syntax will include an adapter *I_Instruction* extending *InterpretableInstruction*. Another instruction is the *HelpCommand*.

An example of the additions made for *Logo* can be seen in Fig. 5. The instructions that correspond to the new entry-points are *Statement* and *Expression*.

4.3 Concrete Syntax Transformation

The second step in our approach is to extend the existing concrete syntax to parse alternatives corresponding to the newly defined instructions. As such, for each instruction *I*, we retrieve the corresponding rule from the base grammar of the DSL and we reuse it for the newly defined adapter *I_Instruction*. The parsing rule *InterpretableInstruction* manages this part.

Another rule is created in order to instantiate an interpreter. Then, the grammar entry-point will be the parsing rule *EntryPoint* that will call either *Interpreter*, *InterpretableInstruction* or *HelpCommand* if the user asks for help on a specific subject.

We also add a custom scope provider in order to resolve the cross references between the previously executed instructions and the current one. When trying to resolve a cross reference, this scope provider will browse through the linked list of the previous instructions. If nothing was found, it will finally turn to the resolution mechanisms of the base grammar.

One of the limitations of this specific implementation is that we use the default Xtext parser to parse single statements. As such, we do not support non context-free grammars. If the language has two semantically different instructions that use the same notation, only one of them can be made into an entry-point. Some possible ways to support non context-free grammars would be:

- to use a custom parser that could build a context from the previously executed instruction (which might add unwanted side effects)
- or to allow the language designer to modify the keywords used by some grammar rules, in order to remove potential conflicts

Fig. 6 depicts an organization of the different artifacts related to the concrete syntax extension of the language *Logo*, while Fig. 7 details the corresponding Xtext production rules.

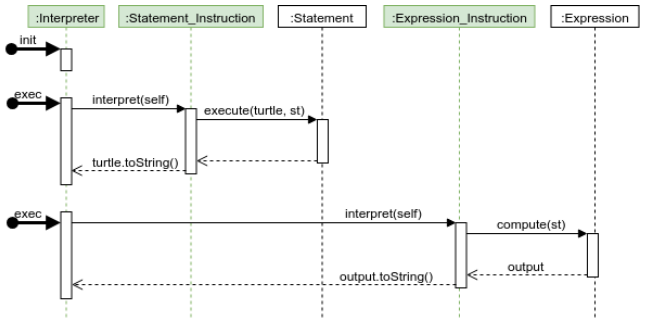

Figure 8. Overall Execution Flow for Logo

4.4 Semantics Transformation

Last, we transform the DSL semantics to incorporate the new execution context and flow management, and to enable the new instructions to be executed.

Here, we handle operational semantics written in ALE. ALE is a language that allows to re-open classes from Ecore metamodels to statically introduce fields and operations at design time. By using the *open class* syntax, we can define the behavior for the classes we added in the syntax, and drive the execution with *@init* and *@main* annotations on operations.

We define the runtime data of the execution context and the initialization of the *Interpreter* entry-point as described in the *ReplDefinition* model. When executed, the interpreter will call the operation `interpret` on the instruction it is currently referencing.

Every instruction adapter takes care of the mapping defined in the *ReplDefinition* model: they become a wrapper that will call the original execution method of the statement or expression, possibly with the right parameters (the interpreter's execution context) and retrieve and display the expected outputs as described in the model.

Fig. 8 describes the overall execution flow for the language *Logo*, while Fig. 9 shows the generated ALE code corresponding to this execution flow. The Interpreter, its initialization method and specialized execution context are derived from the information provided by the language engineer in the *ReplDefinition* model (see section 4.1). For each new entry point, an operation is added to manage the semantics. A new operation is also added to the new *HelpCommand* meta-class.

4.5 REPL Interface and Interactive Environments Examples

Having applied the aforementioned transformation process, the DSL is complemented with a multi entry points parsing of interpretable instructions. In order to build an interactive environments on top of it, we provide a generic REPL interface protocol and its underlying systematic execution framework (cf. Fig. 10):

1. Create an interpreter and initialize it

```
open class Interpreter {
  logolang::Turtle turtle;
  logolang::SymbolTable st;

  @init
  def void init() {
    self.turtle := logolang::Turtle.create();
    self.turtle.xpos := 0.0;
    self.turtle.ypos := 0.0;
    self.turtle.direction := 0.0;
    self.turtle.pendown := false;
    self.turtle.canvas := logolang::Canvas.create();
    self.turtle.canvas.segments := Sequence{};
    self.st := logolang::SymbolTable.create();
    self.st.init();
  }

  @main
  def void run () {
    self.instruction.interpret(self);
  }
}

open class Expression_Instruction {
  def void interpret(Interpreter logo_repl) {
    ecore::EObject output :=
      self.original.compute(logo_repl.st);
    output.toString()?.log();
  }
}

open class Statement_Instruction {
  def void interpret(Interpreter logo_repl) {
    self.original.execute(logo_repl.turtle,
                          logo_repl.st);
    logo_repl.turtle.toString()?.log();
  }
}

open class HelpCommand {
  def void interpret(Interpreter logo_repl) {
    // Call help method of the node
  }
}
```

Figure 9. Generated Extended Operational Semantics for Logo

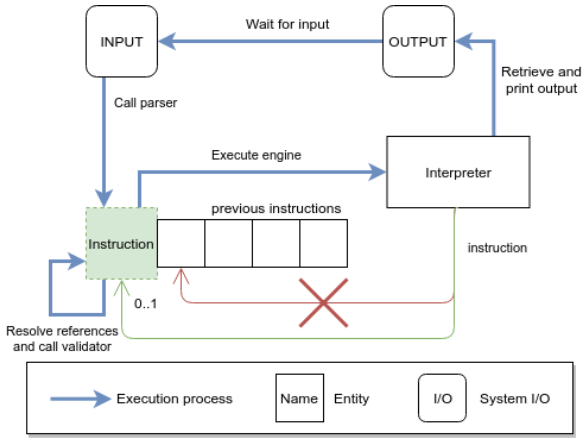

Figure 10. REPL Execution

2. Read the user input
3. Parse it as an instruction
4. Retrieve the previous instruction and store it
5. Swap the instruction of the interpreter for the new one
6. Run the interpreter
7. Print the relevant output
8. Go back to step 2

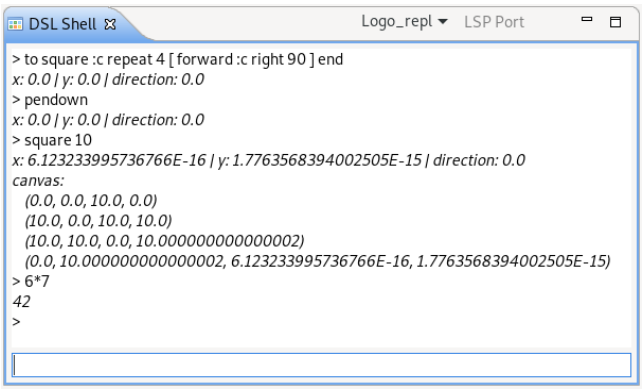

Figure 11. Eclipse Shell Running with Logo REPL

From the new generated DSL specification, we automatically generate the entire GLUE code for integration with two technical environments: Eclipse and Jupyter. For the first one, we provide a plugin including an eclipse view hosting a shell to communicate with the Interpreter. This generic view declares an Eclipse extension point type including among others the name of the REPL language and the qualified name of the interpreter class. Each REPL language declares this extension point. The generic view allows REPL users to select the desired DSL and then start an interactive session. This session keeps track of the executed instructions and offers the ability to reset the interpreter or cancel the last instructions thanks to the environment provided by Gemoc. The Language Server Protocol (LSP[9]) support provided by Xtext enables intelligent completion within the Shell. Figure 11 shows a screenshot of this integration within Eclipse for the Logo language.

For Jupyter, an editor specific to ipynb files (Jupyter notebook) has been created and manually integrated into Jupyter. The purpose of this editor is to replace the default code cell editor (ace editor[10]) with the monaco text editor[11]. The latter has the advantage of natively supporting LSP in order to allow completion, error reporting, etc. A generic glue code has been added to adapt between Jupyter's Kernel concept and GEMOC's execution engine to control the execution of interpreters. Thus for each new REPL, a connection file defining the connection URL to the GEMOC execution engines of this REPL is generated. A descriptor is also generated for Jupyter to register this new kernel.

On the GEMOC side, a class allowing to interface with a ZeroMQ message oriented middleware is created and makes the link with the execution interface of the GEMOC engine and the current REPL. The main advantages of the GEMOC integration is to leverage its execution trace management, debugging facilities and concurrency model management (e.g., to start from any cell or finely control the flow of execution of the cells.).

5 Evaluation

To address the four challenges identified in Section 3, we propose an approach to automatically generate an interactive computer programming environment from a DSL specification and an identification of the execution entry points for this REPL. A first level of validation consists in applying our approach on other DSLs, namely *MiniJava* and *ThingML*, and to reflect on the lessons learnt.

MiniJava is a subset of the general purpose language *Java* that was created for teaching purposes, since *Java* was considered too intimidating for students on various aspects [12]. The first implementation of *MiniJava*, released in 2001, was also shipped with a REPL. This DSL offer a good support to learn Java and test APIs as introduced in Section 2.

We started from an existing implementation in EMF/Xtext/ALE[12]. This specific implementation is a large one, with 80 meta-classes and 200 attributes in the abstract syntax, 170 lines of Xtext for the grammar, and more than 1140 lines of ALE as operational semantics. In order to use real life APIs, we decided to add a support for native *Java* calls through the Java JSR-223 API[13]. JSR-223 is a standard API for calling scripting frameworks in Java. It is available since Java 6 and aims at providing a common framework for calling multiple languages from Java.

We selected nine execution entry-points: Type declarations, Method definitions, Statement blocks, Variable declarations, Assignments, For loops, While loops, Conditions, and Expressions. This means that we added nine `@repl` annotations, including one with a specific output for the expressions. Considering the initial size of the DSL specification, these additions are only nine more lines in the semantics, which can be estimated as a modification of 0.6% to generate a REPL for the existing DSL.

We also added both a Xtext *ScopeProvider*, to manage the scoping, and a Xtext *Validator*, to enforce access rights, to the base definition of *MiniJava*. Having these two new elements written with the pure interpreter pattern inside the DSL definition was not an issue, and they both work as intended for the generated REPL.

The second DSL is an ALE implementation of *ThingML*. ThingML[14] is a domain specific modeling language, that combines well-proven software modeling constructs for the design and implementation of distributed reactive systems:

- statecharts and components (aligned with UML) communicating through asynchronous message passing,

[9] https://langserver.org/
[10] Cf. https://ace.c9.io/
[11] Cf. https://microsoft.github.io/monaco-editor/
[12] Cf. https://github.com/manuelleduc/ale-lang/tree/master/languages/minijava
[13] Cf. https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html
[14] Cf. https://github.com/TelluIoT/ThingML

- an imperative platform-independent action language,
- specific constructs targeting IoT applications.

The ThingML language is supported by a set of tools, which include editors, transformations (*e.g.,* export to UML) and an advanced multi-platform code generation framework, which supports multiple target programming languages (C, Java, Javascript). Recently a simulator has been designed to emulate the distributed system behavior. The abstract syntax contains 88 meta-classes, for a total of 240 model elements. The grammar definition is more than 450 lines long, and writing the operational semantics in ALE require more than 1800 lines.

Being a dataflow language, it was an interesting case study for us since our approach was mainly aimed at imperative DSLs.

In a *ThingML* program, a language user can define types and protocols, with some of those types being "things" that can declare functions and state machines, before instantiating them inside configurations. A configuration of connected things is a dataflow, and the operational semantics of *ThingML* execute it for as long as they have steps to execute.

The syntax of the DSL offers a way to describe complete dataflows, and our approach enables the definition of partial programs. However, the concept of partial dataflows is debatable. Since ThingML uses named elements in a configuration, we could use them as valid execution entry-points, but a smaller granularity would not make sense:

- We could have instantiations and connections between things outside of configurations, but this would mean either that we execute a partial flow from the beginning after each change, or that *ThingML* defines an `execute` instruction (and it does not). Running several configurations is honestly just as good.
- The statements used in functions could be available, but at best they could only be used to interact with completely executed configurations (and their instances if the concrete syntax provided a definition for qualified names), which would be of limited use.
- With the two above, we could actually end up with a complete imperative language, but it would be too far away from the original *ThingML* to stay in the scope of what we want to provide.

As such, we decided to use the following 3 entry-points: Type definition, Protocol definition, and Configuration declaration (and execution). They correspond to 3 more lines in the semantics, which represent 0.12% of the total DSL specification. Our *ThingML* REPL makes it possible to split a program between elements definition and several configuration executions. However, there are a lot more interesting aspects to an interactive environment for a dataflow language: altering an already existing dataflow with new nodes and transitions, or controlling the execution step by step for example.

Lessons Learned This experiment on two DSL specifications defined by other language engineers allowed us to verify several points, and to evaluate our initial RQ and associated four challenges.

The first lesson learned is the ability of our approach to be used on different DSLs as long as these specifications conformed to a certain number of expectations. The language engineer can specify the new entry points of the DSL, the associated outputs and the associated help messages in an expressive way. The effort to define these entry points remains low compared to the level of reuse of the abstract syntax, the grammar and the operational semantics. Such an approach of automated transformation allows a language engineer to have significant confidence in the semantics preservation of the original DSL. Focusing all the tools associated with a DSL only on its specification is an important way to facilitate the evolution of all these artifacts, and in particular the associated REPL. The assumptions made about the form of the implementation of static semantics, the operational semantics, and the different scoping rules are a bit strong. This means that two things are required at the moment: The definition of new entry points must be done by a language engineer and it is required to be able to access the DSL specification in case of issues to correct the parts that do not fully respect the assumption of a pure interpreter design pattern. Finally, if the approach perfectly fits the generation of interactive computer programming environments for imperative languages, many perspectives are opening up in the case of declarative or dataflow languages, and they lead to considering new opportunities for the interactive parts of these kinds of languages.

6 Related Works

Various generic frameworks have been proposed in the last decade to integrate language REPLs, either for education purpose (*e.g.,* Repl.it[15]) or scientific computing (*e.g.,* Jupyther[16]). In all cases, a specific implementation of the language must be provided. While the implementation is time consuming, this is also error prone and needs to be aligned with the initial semantics of the language.

Bacatá has been recently proposed to automatically derive a new kernel for Jupyter from a DSL specification defined within the language workbench Rascal [10]. While all the implementation is automatically generated from the specification, the specification (i.e., the syntax and the semantics) has to be defined specifically for supporting a REPL.

Our approach automates the transformation from an initial specification for a textual interpreted DSL to a new specification and the underlying language implementation for being integrated into an interactive computer programming

[15] cf. https://repl.it
[16] cf. https://jupyter.org

environnement. The resulting implementation can be integrated into either a simple language shell or more complex environments such as Notebooks. To the best of our knowledge, there is no related work addressing the specific challenge of automatically transforming a language syntax and semantics to support interactive programming (*i.e.*, multiple execution entry-points, and management of the execution context and flow).

7 Conclusion and Future Work

We describe in this paper an approach to automatically transform an existing specification of a textual and interpreted DSL, into a new specification that drives the development of an interactive computer programming environment. From additional information about the allowed entry points and the expected outputs when executed, we describe how to transform the grammar specification and the operational semantics specification so that we can have multiple execution entry points, and a sound and extensible management of the execution context and flow. We also define a unified interface to be used from different interactive environments such as a language shell and a notebook interface. The implementation and the evaluation have been done in the GEMOC Studio, but the proposed approach could be implemented in other language workbenches.

This approach opens up various perspectives. While our approach is currently expecting operational semantics in the form of an interpreter, we would like to extend it in the future to also cover translational semantics in the form of a compiler. We would also investigate the support of a seamless interoperability [4] between the interactive computer programming environments and the initial environment. In the long term, we would like to investigate polyglot interactive environments offering a seamless integration of heterogeneous languages.

References

[1] Mathieu Acher, Benoit Combemale, and Philippe Collet. 2014. Metamorphic Domain-Specific Languages: A Journey Into the Shapes of a Language. In *Onward! Essays*. ACM, Portland, United States, 243–253. https://doi.org/10.1145/2661136.2661159

[2] Eric Allen, Robert Cartwright, and Brian Stoler. 2002. DrJava: A Lightweight Pedagogic Environment for Java. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*. ACM, New York, NY, USA, 137–141. https://doi.org/10.1145/563340.563395

[3] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. Amsterdam, Netherlands, 8. https://hal.inria.fr/hal-01355391

[4] Fabien Coulon, Thomas Degueule, Tijs Van Der Storm, and Benoit Combemale. 2018. Shape-Diverse DSLs: Languages without Borders (Vision Paper). In *SLE 2018 - 11th ACM SGIPLAN International Conference on Software Language Engineering*. ACM, Boston, United States, 215–219. https://doi.org/10.1145/3276604.3276623

[5] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[6] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1997. DrScheme: A pedagogic programming environment for scheme. In *Programming Languages: Implementations, Logics, and Programs*, Hugh Glaser, Pieter Hartel, and Herbert Kuchen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–388. https://doi.org/10.1007/BFb0033856

[7] Kathryn E. Gray and Matthew Flatt. 2003. ProfessorJ: A Gradual Introduction to Java Through Language Levels. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. ACM, New York, NY, USA, 170–177. https://doi.org/10.1145/949344.949394

[8] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS'16)*. ACM, New York, NY, USA, 125–135. https://doi.org/10.1145/2976767.2976812

[9] Jeffrey M. Perkel. 2019. Pioneering 'live-code' article allows scientists to play with each other's results. *Nature* 567 (02 2019). https://doi.org/10.1038/d41586-019-00724-7

[10] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. 2018. Bacatá: A Language Parametric Notebook Generator (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2018)*. ACM, New York, NY, USA, 210–214. https://doi.org/10.1145/3276604.3276981

[11] F. Perez and B. E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science Engineering* 9, 3 (May 2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

[12] Eric Roberts. 2001. An Overview of MiniJava. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE'01)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/364447.364525

[13] Uri Sarid. 2016. API notebook tool. US Patent 9,442,700.

[14] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature* 515 (11 2014), 151–2. https://doi.org/10.1038/515151a

[15] Andrew Sorensen. 2005. Impromptu : an interactive programming environment for composition and performance. In *Australasian Computer Music Conference 2009 : Improvise*. Queensland University of Technology, Brisbane. https://eprints.qut.edu.au/31056/

[16] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[17] Christophe Tombelle and Gilles Vanwormhoudt. 2006. Dynamic and Generic Manipulation of Models: From Introspection to Scripting. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings (Lecture Notes in Computer Science)*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.), Vol. 4199. Springer, 395–409. https://doi.org/10.1007/11880240_28